Decentralized Deep Learning with Gradient Compression

Cui, Xinzhe

(出版者 / Publisher)
法政大学大学院情報科学研究科
(雑誌名 / Journal or Publication Title)
法政大学大学院紀要. 情報科学研究科編
(巻 / Volume)
14
(開始ページ / Start Page)
1
(終了ページ / End Page)
6
(発行年 / Year)
2019-03-31
(URL)
https://doi.org/10.15002/00021917

Decentralized Deep Learning with Gradient Compression

Cui Xinzhe
Graduate School of Computer and Information Sciences
Hosei University
xinzhe.cui.34@stu.hosei.ac.jp

***Abstract*—Training a deep neural network (DNN) with a single machine consumes much time. To accelerate the training, a popular method is distributed deep learning (DDL), that is to splits and distributes DNN training tasks on multiple machines. One of the common designs of the DDL is a centralized architecture, that contains one parameter server and several workers. In this architecture, several workers need communicate with the parameter server to get the latest parameter at every iteration. The network resource of the parameter server will be exhausted frequently, which further lead to the limitation of the total training performance. In this paper, we introduce a decentralized approach for distributed deep learning. Our proposed decentralized DDL removes the parameter server and all worker nodes are designed to communicate with each other directly to exchange the intermediate results of their own model. For good generalization, we built a customized optimizer class to compress the every iteration's gradient that only update the gradient larger than threshold value. Compared with centralized distributed approach, our proposed model gains better accuracy during a whole test. When we limit the parameter server's bandwidth to 300Mbps in order to simulate the network bottleneck problem, the performance of our proposed decentralized DDL improved several times faster than the existing centralized approach on the worst case execution time.**

Keywords—Deep Learning, Distributed Computing, RPC, Decentralized Algorithm , Gradient Descent, TensorFlow

I. INTRODUCTION

Deep learning methods have shown amazing accuracy in areas such as conversation recognition, image recognition, object detection, and drug discovery and genomics, and so deep learning is also used in more and more practical businesses. Training the model using the newly generated data is important to improve the accuracy of the application. Therefore, it is important to increase the training speed of the model. In most cases, it may take several weeks or even months if the model is trained on a single machine. Distributed deep learning, which uses multiple devices to train the model in parallel, will accelerate the process of the training convergence.

Early distributed deep learning training was performed using a distributed machine learning framework such as PyTorch, Caffe, Petuum. Many following series of distributed platforms dedicated to deep learning tasks have emerged, such as DistBelief, Tensorflow, MXNet, Adam, and others. These mainstream distributed deep learning architecture requires one or more parameter servers(PS) as the central node [1], and the computing nodes continuously communicate with the central node for parameter transfer and update. Therefore, network communication will become a serious bottleneck to limit the cluster's performance. This means that when we use tens or hundreds of nodes to train the model, most of the time in the cluster is wasted on node communication.

In this paper, we design and implement a decentralized distributed deep learning architecture based on TensorFlow and Remote Procedure Call(RPC). Each computer of the cluster in the same level. There are only peer nodes in the cluster and there are no parameter servers and workers. Each node will select a peer node to communicate and deliver the model parameters. After each node receives the parameter data from the model sent by the neighbor node, it will fuse with the model parameters of the local node and take the merged parameter as the next iteration's training parameters.

II. RELATED WORK

A. Distributed Deep Learning with Parameter Server

Distributed parallel iterations can be divided into synchronous and asynchronous distributed deep learning according to the model update mode. The similarities between these two distributed methods are that there are one or more parameter servers and one or more workers on the node allocation. The main function of the parameter server is to collect gradients and update parameters from the worker node. The task of the worker node is to calculate the gradient and obtain the updated model parameters from the parameter server, and then perform the next round of iterative training [12]. The difference between these two distributed methods is the update strategy:

Synchronous approach: At each iteration of synchronous distributed deep learning, the parameter server waits until all worker nodes in the cluster complete the current iteration and collects all parameters[2]. Then the parameter server updates the local model. After the worker node get the latest model from the parameter server, the next training iteration will be executed. If a worker node in the cluster is faulty or the calculation is slow, a fallout occurs. All nodes in the cluster, including the parameter server and the worker node, wait for the dropout to complete the task before continuing. Therefore, when the network is unstable or abnormal conditions such as unstable nodes in the cluster occur, all the nodes in the cluster must wait for the node to complete the task [5]. Moreover, since the parameter server needs to collect the calculation results of multiple worker nodes, the network resources of the parameter server also become limited task iterations, which makes the clusters often idle, resulting in waste of computing resources [6].

Asynchronous approach: The asynchronous distributed deep learning is when each worker node communicates with the parameter server immediately after completing the current iteration, submits the calculation result, the parameter server updates the model according to the submitted calculation

result, and then the worker node immediately executes the updated model. The next iteration. This avoids the situation where the nodes in the cluster wait for the stragglers. However, the disadvantage of asynchronous distributed learning is the presence of staleness [2][4][7]. As shown in the Fig. 1, when both nodes are near point A, worker1 and worker2 calculate the gradient δ at the same time so that the model can reach the minimum point O. But while worker1 has updated the model which stores on the parameter server, worker2 don't know the model has updated. Then the worker2 will update the model once again so that the model reaches point B, so the convergence speed of asynchronous distributed learning will be slower, and the accuracy sometimes has a little loss, but as the number of iterations increases, the model will eventually converge to a satisfactory result, and the asynchronous distributed learning approach has lower requirements on the stability of the cluster. Even if individual nodes fail, it will not affect the training of the model.

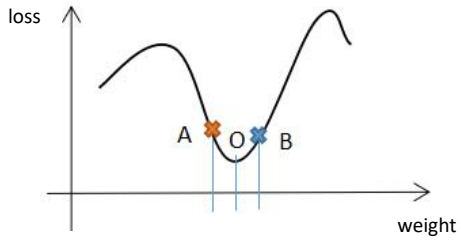

Fig.1. Staleness Problem. We want to optimize the function of loss from point A to O. And we can use the gradient descent algorithm to find the right weights vector which at point O corresponding position of axis weight.

B. Tree-AllReduce

This method is to choose a GPU as the root node and other GPUs as children to build the communication structure [9]. The root node is responsible for collecting the results of child node calculations and collecting the collected data. Each round of training iterations requires that all cards complete the data once and then reduce it. If the number of cards is relatively small, the effect is insignificant. However, if there are many cards in parallel, it involves a situation in which a fast calculation card needs to wait for a slow card to be calculated, resulting in waste of computing resources. Each iteration of all computing GPU cards needs to communicate with the root card for all model parameters. If the data volume of the parameters is large, the root card's communication overhead is also become very large, and the linear growth overhead will limit the GPU cluster's performance.

C. Ring-AllReduce

This method builds the GPU cards as a ring. While the GPU card communicates in a ring, each card has a left-handed card and a right-handed card. Because the network structure on each card is fixed, the parameters in the card are the same. During each communication, just send the parameter to the right-hand card, and then the card receives data from the left-hand side. After repeated iterations, synchronization of the entire parameter is achieved. However, this method is effective under the GPU cluster. Under multi-machine nodes, the instability of the network and the transmission of the network are much smaller than the throughput of the graphics card, which easily leads to the waste of GPU resources.

III. DECENTRALIZED ALGORITHM

A. Algorithm Design

[3] has proposed the decentralized gradient optimization algorithm. It use the averaged gradient to implement the decentralized algorithm. Inspired by this approach, we averaged the model instead of the gradient. The peer-to-peer deep learning algorithm is described as follows , Algorithm 1 shows the complete algorithm. Every node has a local model, the model will be trained on every node dependently, and every node will repeat the following steps:

Algorithm 1 Peer-to-Peer Deep Learning

Input: learning rate γ
Input: the total number of iterations T
Input: shared variable between processings *shared_vars*
Output: local model's parameters x_T^i

function SERVERING(*shared_vars*)
  Listening the fixed port. If receive peer's request, return the local model's weights
  if *Requested* **then**
    $shared\_vars \leftarrow x_t^i$
    send the local model's parameters *shared_vars* to peer node
  end if
end function

function TRAINING(*shared_vars*)
  Randomly initial the model's parameters x_0^i
  $shared\_vars \leftarrow x_0^i$
  for $t = 0, 1, 2, \cdots T$ **do**
    Randomly sample dataset ξ_t^i from local dataset
    Compute the stochastic gradient $\nabla F_t(x_t^i, \xi_t^i)$ subject to the ξ_t^i and x_t^i
    $threshold \leftarrow mean(\nabla F_t(\xi_t^i, x_t^i))$
    $\nabla F_t(\xi_t^i, x_t^i) \leftarrow \nabla F_t(\xi_t^i, x_t^i) \geq threshold$
    $x_{t+1}^i \leftarrow x_t^i - \gamma \nabla F_t(\xi_t^i, x_t^i)$
    Each node will randomly choose a peer node to request its weights x_t^j
    if *Getted* : **then**
      $x_{t+1}^i \leftarrow \frac{1}{2}(x_{t+1}^i + x_{t+1}^j)$
    end if
    $shared\_vars \leftarrow x_{t+1}^i$
  end for
end function

1. **Sample dataset:** Every node will randomly sample a mini-batch dataset which dataset size is m from training dataset denoted by ξ_t^i ,where i is the i-th node, t is the iteration number.
2. **Compute gradient:** According to the sampled dataset ξ_t^i , compute the gradient $\nabla F\ x_t^i\ \xi_t^i$ subject to x_t^i where i is the i-th node, t is the iteration number, x_t^i means the set of local model's parameter tensor.
3. **Compute the threshold of every layer's gradient:** Compute every layer's gradient mean value as the threshold. And according to the threshold value make masks.
4. **Compress the gradient:** The gradients which greater than the mean value will be updated, and less than mean value will be dropped out.
5. **Update local model:** Update the local model's parameters $x_{t+1}^i \leftarrow x_t^i - \gamma \nabla F\ x_t^i\ \zeta_t^i$, where γ is the step size.
6. **Get peer node's model:** Use RPC client connect other peer's RPC server, and request their model's parameters denoted by x_t^j .
7. **Average model:** If the client get the peer's model parameters, then the local node will average the local

model with the peer's model $x_{t+1}^{i} \leftarrow \frac{1}{2} x_{t+1}^{i} + x_{t+1}^{j}$, and assign the averaged model as the next training iteration's parameter. If any exceptions occurred, the local node will ignored this step, and use the updated local model x_{t+1}^{i} as the next training iteration's parameter.

8. **Assign averaged model to local model:** Before we assign the model, we need define the placeholder in the computing graph. Then use the dictionary feed into the graph and start the next training iteration.

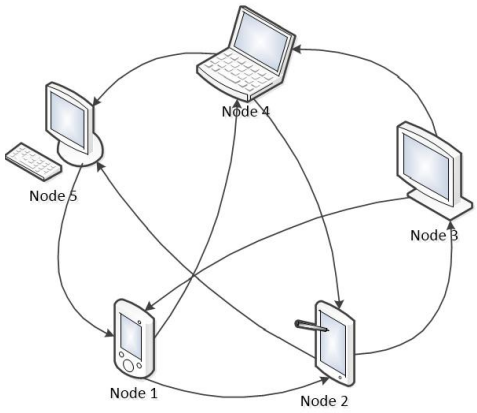

Fig. 2. Mesh communication. Through the decentralized deep learning, various of devices can communicate with each other devices locally, without sending user data to the server.

According to the algorithm description, the distributed deep learning architecture in this research is implemented as displayed in Fig. 3. In this distributed training approach, every node in the same level, without centralized node, and only peer node in this cluster. Every node can communicate with each node directly, just like human talking face to face. This kind of architecture aim to make the smart device learning with other devices through peer to peer communication approach. This approach can protect user's private data because of only model parameter will be transmitted between nodes and avoid the network resource bottleneck on the centralized node. This peer to peer communication approach make the nodes connected like a mesh as shown in Fig 2.

B. Architecture Implemention

While implement this architecture like Fig. 3, I use the multiprocessing to build the models because of the Python language has the Global Interpreter Lock(GIL). When the node launches the training job, it will start two processes. One processing is the RPC server processing, it will listen the fixed port continuously to response other peer node's request and return the local model's parameters to the node which send the request. The other processing is responsible for the training job, it will be responsible for sample dataset, compute the gradient, compress the gradient, get peer node's model, and update the local model.

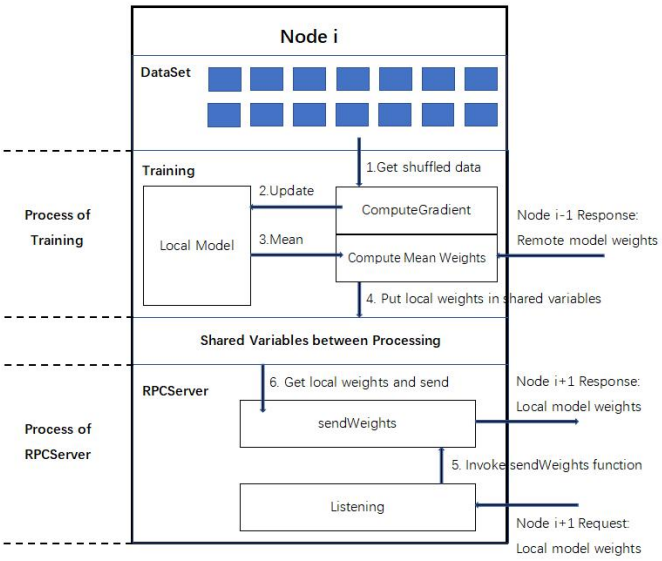

Fig.3. Peer-to-Peer Deep Learning Architecture. In this architecture, we use multiprocessing to complete every node's training job. One processing for the training model, and the other one for listening. The training processing can calculate the latest model, the listening processing can send the latest model to peer node. Between the processes, there has a shared variable, which can be accessed by two processes.

Because of the asynchronous will incur the staleness problem we set a threshold to remove part of the small gradient. Every iteration only update part of the important gradient will reduce the probability of staleness problems. So here I choose a mean value as the threshold, the less than the threshold value will be dropped out, in this approach we can reduce the impact of staleness, in the same time we can get a suitable convergence rate.

When we contact with the multiprocessing problems, we need to consider whether the deadlock will be occurred in our system. It is a very common problem, because of competing resources or communicating with each other. In our system, we set the exception handling mechanism, while the client request someone's model parameters, but no response from the server, after a certain time, the client will give up this request, and go to the next iteration training. Similarly, the exception during parameters transfer also make the system go to the next iteration training without any system crash risk.

Because of every node in the same level, so the communication topology has many choices. The simple and useful choice is Ring topology. It can exchange the parameters between the peers easily and efficiently. And the peer node also can randomly choose a peer to communicate, but this randomly approach may lead many peer communications with same peer, so a pre-designed and reasonable communication rule can improve communication efficiency and avoid the appearance of similar problems.

The final aspect is about the average strategies. While we can simply use the mean approach to merge local model and peer's model. But when some node is seriously behind other nodes, we also use the mean strategy will slow down the fast nodes, so we use the contribution rate to judge which model will account for a greater proportion. If two models has similar loss, this strategy is same as the mean strategy, but

when some node has got a much lower loss, but another node just start training job, the better model will much important than the initial model. In this strategy, no matter we start the training job at the same time or not, the node can dynamically change the model's proportion to get a better convergence rate.

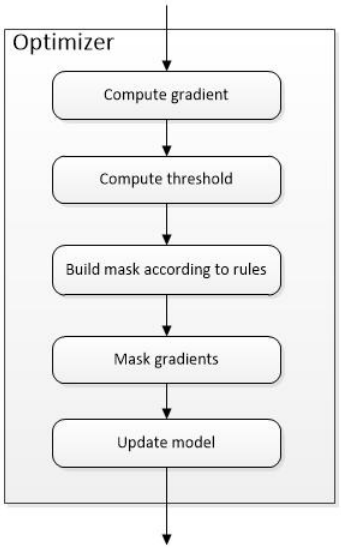

Fig.4. Customized Optimizer. In order to avoid adding extra calculations, I have to overwrite the optimizer class of TensorFlow. I add some features to process the gradient in the customized optimizer like the figure shows.

During implement the simulation demo, we use the TensorFlow framework, because this framework has a lot of resources about the source code. Cause of we need better performance, so we cannot add additional computing task as possible as we can, the only choice we can choose is use the TensorFlow because of many basic functions we can easily overwrite and invoke rather than the Keras which has friendlier API.

When we program the gradient compression model, we need compute every layer's gradient and then we need compute every layer's threshold, because every layer has different shape, and the values also on the different magnitude, so the threshold should be calculated independently. At first, we implement this part after the gradient computing step, this approach will lead redundant calculations so that every step will cost roughly 0.6 more second. That is because the program need retraverse every layer's gradient to calculate the threshold.

To avoid the retraverse, we overwrite the optimizer's source code. We inherit the basic optimizer class and add the threshold computing operation and the mask operation as shown in Fig. 4. In this way, every training step will get the masked gradient, and update the model directly. This solution compares with the original training speed without any extra time consuming.

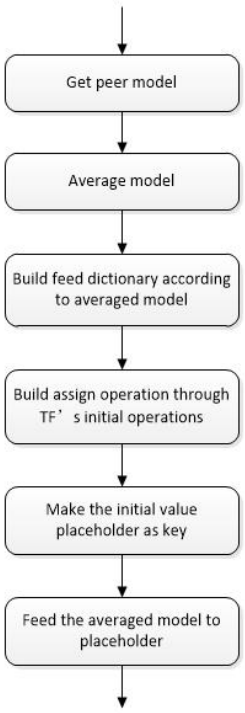

Fig.5. Assignment Operation Definition. While we calculate the averaged model's parameters, we need to assign it to the local model. Also considered the performance, we borrow the initial operation which is native operation from framework to serve our assigned operation.

Fig. 5 shows another part is assign the averaged model to the local neural network. While we use TensorFlow to build our model, one of this framework's character is graph computing. If we want to assign the averaged model to local model, we need define the computing operation in the graph before we start the training job [8]. And we build the initial operation according to the TensorFlow's built-in features which is add "/Assign" string after the variable names as the initial operation name. And then we can use the initial values placeholder as the averaged model parameters' key to build feed dictionary. Finally, we can easily and efficiently feed the averaged model to the local model during we execute the computing graph.

IV. Evaluation

We performed experiment based on 5 machines. These machines have the same configuration as follows: the CPU is Intel® Xeon® Processor X3450, the memory size is 16GB, the operating system is Ubuntu 14.04 LTS, the bandwidth is 300M/bps. And we use Cifar10 as the evaluation dataset, the neural network structure is VGG-19, the optimizer is Momentum optimizer. In the cluster, there has 4 nodes as worker, and 1 nodes as parameter server. During experiment, we set the parameter server's bandwidth to 300Mbps, because of the 4 nodes cannot reach the parameter server's network bottleneck if the nodes communicate through the local network. And while training the model through the asynchronous approach, we statistic the training speed for every training batch. As the histogram shows, about 10000 batches training speed cost less than 0.7 second every batch, and about 3/4 training batch cost more than 0.7 second. The distribution of the speed is showed on Fig. 6.

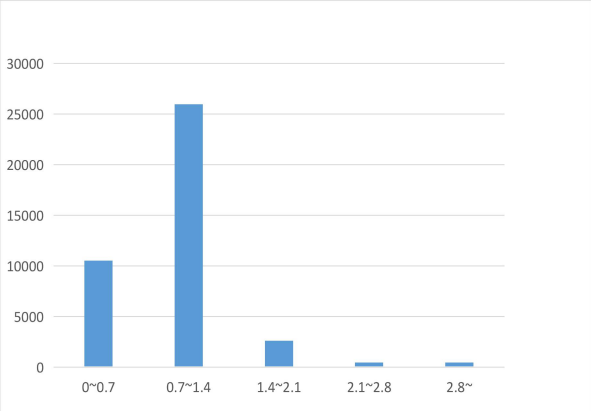

Fig.6. Asynchronous Training with PS Speed Histogram

First, we compare the time consuming between decentralized and centralized approach. As the Table I shows, the centralized approach need about 1.06x to 5.81x time compare with decentralized approach. Fig. 7 shows the result without PS. Because of the network cannot transmit so many data, many jobs will be put on the task queue. So, the centralized speed has a large fluctuation. But the decentralized approach training speed is stable since every node only has one download and one upload task at every moment.

TABLE I. TIME CONSUMING

Decentralized	Centralized
0.58sec/batch~0.73sec/batch	0.62sec/batch~3.43sec/batch

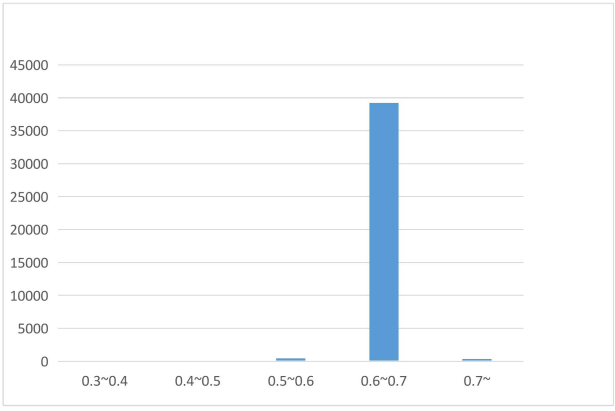

Fig.7. Asynchronous Training without PS Speed Histogram

Then the following Fig. 8 shows the curve of loss while training the model. The training iteration is 40000 totally. The four nodes training model dependently without parameter server, so every node has different model and the curve shows the convergence process of loss. The final loss of node1 is 0.3670, node2 is 0.2103, node3 is 0.1774, node4 is 0.1688. The Fig. 9 shows the centralized distributed approach's loss curve. After 40000 training iterations, we got the loss is 0.4877. The Fig. 10 shows the model's loss curve which training on single machine, the final loss is 0.4192.

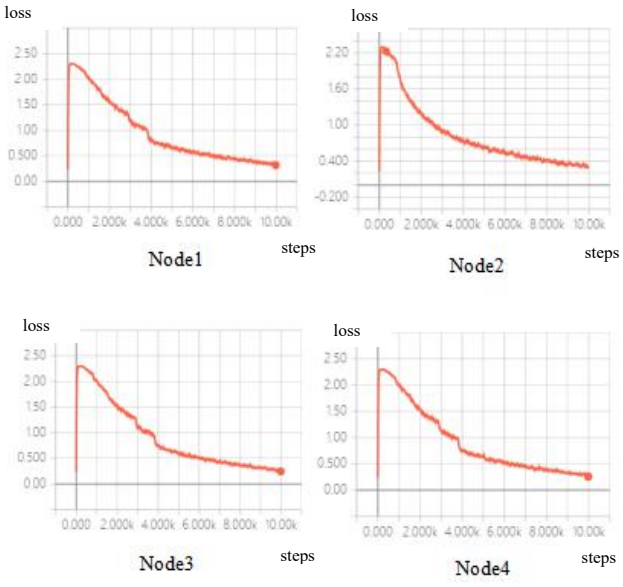

Fig.8. Every node's result with Decentralized Approach

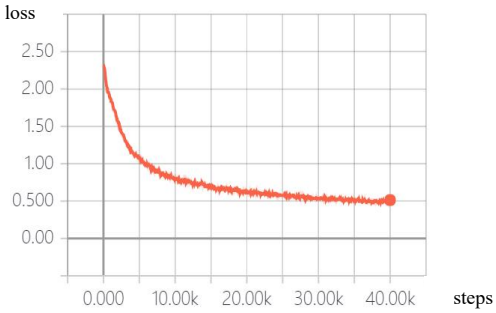

Fig.9. 4 Workers and 1 PS with Centralized Approach

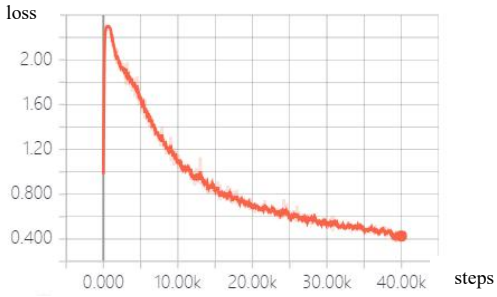

Fig.10. Single-machine Approach

The following Table II shows the training accuracy. As the table shows, the training accuracy of decentralized approach are 86.7%, 92.2%, 95.3%, 94.5%. Through the centralized distributed way, the final training accuracy is 96.7%. The training accuracy which trained on single machine is 96.5%.

TABLE II. TRAINING ACCURACY

Decentralized	Node1	Node2	Node3	Node4
	86.7%	92.2%	95.3%	94.5%
Centralized	96.7%			
Single Machine	96.5%			

Finally, Table III shows the test accuracy which is the most useful evaluation indicator. As we can see, through decentralized approach we got 4 models, while we test the accuracy use the test dataset, they got the results are 79.9%, 79.6%, 79.8%, 79.8%. Through the centralized distributed way, the final test accuracy is 79.5%. The model which trained on single machine got 83.8% accuracy.

TABLE III. TESTING ACCURACY

Decentralized	Node1	Node2	Node3	Node4
	79.9%	79.6%	79.8%	79.8%
Centralized	79.5%			
Single Machine	83.8%			

V. DISCUSSION

According to the evaluation's results, we can see that the decentralized distributed deep learning approach have different training accuracy during training stage. And the centralized approach's training approach also has a better accuracy than the decentralized result.

When we use the test dataset to evaluate the model, the decentralized model can get a similar test accuracy after 40000 iterations without any fine tuning to the centralized distributed results. Although every model has very different training accuracy, but their test accuracy is all near 80%. And we can get a good model although every model is independently train 10000 iterations. So, this decentralized approach to train the neural network can converge as fast as the centralized approach and without the parameter server the asynchronous distributed approach also can get a good model which has similar accuracy.

Compared with the existing centralized approach, the decentralized proposal is more complexity and scalability, because of the network bottleneck on the parameter server has been removed. Every node can join the cluster at any time because we can dynamic adjust the percentage of the local model and peer model according to the model's contribution of loss. Without parameter server, the cluster can get the similar accuracy. But the weak point is that we don't has a mechanism which for maintaining the node's contacts. That means we should set the node's IP before start training if a new node adds into the cluster, others don't knows the node's IP address. This will be a part of our future job. In the same time, this decentralized distributed deep learning architecture not only avoids the communication bottleneck problem of the existing centralized architecture parameter server, but also increases the privacy of personal data because devices don't need send the data to the server. When smart devices come into our lives in a wide range, the way in which smart devices exchange data directly between them can greatly improve the flexibility and scalability of distributed learning. Smart devices can also be based on users' own its customary to train more personalized models to enhance the user's experience. In the future, many small smart cluster can locally train their own model, the decentralized approach will be much suitable for this scenario.

VI. CONCLUSION

In this paper, we implement the neural network in different ways: single-machine version, asynchronous with parameter server, asynchronous without parameter server. Then we compare the different approaches performance without any fine-tuning work. The single-machine version gets the best test accuracy is 83.8%, but the whole training processing is time consuming. Centralized and decentralized approach get similar test accuracy while every node training 10000 iterations. So, the decentralized deep learning can make the node train model through peer to peer approach. In this way, we can avoid the parameter server's network bottleneck without test accuracy loss.

REFERENCES

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard et al., "Tensorflow: A system for large-scale machine learning," arXiv, pp. 1605, 2016.

[2] Wei Zhang, Suyog Gupta, Xiangru Lian, and Ji Liu, "Staleness-aware Async-SGD for distributed deep learning," IJCAI, 2016.

[3] Mokhtari and A. Ribeiro, "Decentralized double stochastic averaging gradient," In Signals, Systems and Computers, 2015 49th Asilomar Conference on, pp. 406–410, IEEE, 2015.

[4] Defazio, F. Bach, and S. Lacoste-Julien, "Saga: A fast incremental gradient method with support for non-strongly convex composite objectives," In Advances in neural information processing systems, pp.1646–1654, 2014.

[5] Shahrampour and A. Jadbabaie, "Distributed online optimization in dynamic environments using mirror descent," IEEE Transactions on Automatic Control, 2017.

[6] A. T. Suresh, F. X. Yu, S. Kumar, and H. B. McMahan, "Distributed mean estimation with limited communication," Proceedings of the 34th International Conference on Machine Learning, vol. 70, pp. 3329–3337, 2017.

[7] O. Dekel, R. Gilad-Bachrach, O. Shamir, and L. Xiao, "Optimal distributed online prediction using minibatches," Journal of Machine Learning Research, 2012.

[8] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, et al, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," arXiv preprint arXiv, pp. 1603.04467, 2016.

[9] P. Patarasuk and X. Yuan, "Bandwidth optimal all-reduce algorithms for clusters of workstations," Journal of Parallel and Distributed Computing, 2009.

[10] P. Bianchi, G. Fort, and W. Hachem, "Performance of a distributed stochastic approximation algorithm," IEEE Transactions on Information Theory, 2013.

[11] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," 2009.

[12] X. Zhang, J. Zou, K. He, and J. Sun, "Accelerating very deep convolutional networks for classification and detection," IEEE transactions on pattern analysis and machine intelligence, vol. 38, pp. 1943–1955, 2016.

[13] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," arXiv preprint arXiv, pp.1409-1556, 2014.